PipeStreams: Pipelines and Streams

Terminology

■ **Sources** (a.k.a. 'readables') do not take inputs from the stream, but provide data to 'flow downstream'. Sources may be constructed from values (e.g. a text may be converted into a source of text lines or a series of characters), from files (reading files and processing their contents being a frequent application of streams), or using functions that sit around until they are called with pieces of data they then send downstream.

■ **Sinks** (a.k.a. 'writables') do not provide outputs to the stream, but accept all data that is funnelled to them by way of the pipeline. There are two important subclasses of sinks: on the one hand, sinks may be used to write to a file, an outgoing HTTP connection, or a database; so that would be sinks with targets. Target-less sinks may seem pointless but they are needed to provide a writable endpoint to a pipeline. The function used to represent such a sink is known as `$drain` in PipeStreams, so 'a drain' just means that, a generic target-less sink required by the API.

■ **Pipelines** are lists of **stream transforms**. As we will shortly see, pipelines represent the central constructional element in the PipeStreams way of doing things. They start out as generic lists (i.e. Javascript `Array`s) that transforms—functions—are being inserted to, and they end up representing, in the ordering of their elements, the ordering of transformational steps that each piece of data fed into their top end has to undergo before coming out at the other end. A well-written pipeline combines a simple conceptual model with a highly readable list of things to do, with each station along the assembly line doing just one specific thing.

■ **Transforms** (symbolized as `$f()`) are synchronous or asynchronous functions that accept one data item d_i (a.k.a. events) from upstream and pass zero or more data items d_1, d_2, … down the stream, consecutively.
In short one can say that in each stream, data comes out of a source, flows through a number of transforms $\$f_i$, and goes into some kind of sink. A **complete pipeline** has at least a source and a sink.

■ By **streams** (symbolized as Σ) we mean the activity that occurs when a complete pipeline has been 'activated', that is made start to process data. So, technically, a 'stream' (a composite algorithm at work) is what a 'pipeline', once activated, does. In practice, the distinction is often blurred, and one can, for example, just as well say that a particular event is 'coming down the stream' or 'coming down the pipeline'.

Equivalence Rules

The power of streams—and by streams I mean primarily `pull-stream`[1]s on which PipeStreams is built—comes from the abstractions they provide. In programming, *functions* are such powerful abstractions because when you take a chunk of code and make it a function with a name and a call signature, all of a sudden you have not only a piece of code that you can pass around and invoke, you can also put that invocation into *another* named function and so on. So, a building block made from smaller building blocks remains a building block, albeit a more complex one.

Likewise, when building processing pipelines from stream transforms, you start out with pipelines built from stream primitives, and then you can go and put entire pipelines into other pipelines, taking advantage of the compositional powers afforded by the equivalence rules (invariants) that streams guarantee. In the below, we use an arrow `a -> b` to symbolize 'a is equivalent to b', i.e. 'b acts like an a'. `pull` represents the PipeStreams `pull` method (basically `pull-stream`'s `pull` method); `$f()`, `$g()` represent stream transforms (see *API Naming and Conventions* for the leading dollar sign); the ellipsis notation `$f(), …` represents 'any number of transforms'.

■ A pipeline with a source and any number of transforms is equivalent to a source:

`pull [ source_1, $f(), …, ] -> source`

■ A pipeline with any number of transforms and a sink is equivalent to a sink:
`pull [ $f(), …, sink_1 ] -> sink`

■ A pipeline with any number of transforms is equivalent to a (more complex) transform:
`pull [ $f(), …, ] -> $g()`

■ In particular, a pipeline with no elements is equivalent to the empty (no-op) transform that passes all data through (and that can be omitted in any pipeline except for a smallish penalty in performance):
`pull [] -> PS.$pass()`

■ A pipeline with a source, any number of transforms and a sink is equivalent to a stream:
`pull [ source, $f(), …, sink, ] -> stream`

API Naming and Conventions

`PS.pull pipeline…`

Simple Examples

Ex. 1

```
PS  = require 'pipestreams'
log = console.log
p   = []
p.push PS.new_value_source [ 'foo', 'bar', 'baz', ]
p.push PS.$show()
p.push PS.$drain -> log 'done'
PS.pull p…
```

Ex. 2

```
PS              = require 'pipestreams'
{ $, $async, }  = PS

$double = ->
  return $ ( d, send ) -> send 2 * d

source  = PS.new_push_source()
p       = []
p.push source
p.push $double()
p.push PS.$show()
p.push PS.$drain()
PS.pull p…
source.push 42
```

Comparison with NodeJS Streams, Pull-Streams

Here are a few points that highlight the reasons why I wrote the PipeStreams library on top of pull-stream[1]s (after writing PipeDreams[2] which were built on top of NodeJS Streams[3]):

■ The basic API ideas of PipeDreams[4] turned out to be a highly useful and effective tool to create not-so-small data processing assemblies. Before pipelines, such assemblies tended to be ad-hoc messes of synchronous and asynchronous pieces of code calling each other; after pipelines, assemblies could be written as linear sequences of named functions.

■ The PipeDreams stream transform call convention—where a transform is (produced from) a function `( data, send ) ->` that accepts a piece of `data` and a `send` method that is used to send data downstream—proved to be the main enabling aspect of said library. All of a sudden you could just dump all that is wrong with NodeJS streams[5] and forget about all their Byzantine complexities[6]: just write a function that `( data, send ) -> … send data …` and bang, you're good to go.

■ PipeDreams had some downsides, though; apart from some of the *complexities* of NodeJS streams that could not be entirely hidden, it also suffered from their *inherently mediocre performance characteristics*[7]: the architecture of NodeJS streams is such that adding a transform to a pipeline incurs a non-trivial run-time performance penalty, so much that **the performance of NodejS streams pipelines with more than a very few steps will be dominated by the number of steps, even if those steps are no-ops**; this at least used to be the case at the time when I abandoned NodeJS streams and turned to Pull-Streams. The whole idea of streams is to do one little thing at a time and have those many little steps co-operate to accomplish a bigger goal; an implementation with an unreasonable cost on adding steps ruins that picture.

■ The underlying implementation of Pull-Streams is hugely simpler[8] than that of NodeJS streams. To quote another guy[9] who thinks so:

> pull streams' # 1 superpower is their sim-

> plicity (in the Rich Hickey sense of the word): anyone can write a full pull stream implementation from scratch in a few minutes, from first principles. This is not true of node streams in the slightest. Simplicity brings transparency with it, meaning debugging and reasoning about implementation gets easier.

So while 'simple' doesn't equal 'easy' (in the Rich Hickey[10] sense of the word) it's still true that simpler concepts, a simpler implementation and a simpler API are to be preferred over a convoluted implementation (and API) that suffers from backward-compatibility pressures and maintains several parallel, mutually exclusive and ultimately superfluous modes of operation. In the case of NodeJS streams, you have 'new style' vs 'old style' mode of operation: A switch that is done transparently based on what seemingly unrelated parts of the API you employ in what order. Next, you must decide whether you're dealing with 'objects' or 'binary' data, a completely gratuitous difference: it's just data. Lastly, you can structure your streaming app to do things the `.pipe()`ing way, or, alternatively, the `EventEmitter` way—inheriting all that is wrong with the `EventEmitter` API and implementation[11]. To top it off, you still don't get proper error handling with NodeJS streams[12].

> I'm not saying that using the event handling model to process streams is wrong, I just claim that having NodeJS do both piping and events where either would have sufficed is what contributes to the rather sad performance story they deliver.

I'm really sorry that these points amount to what can be perceived as bashing on the NodeJS folks who have given us the great piece of software that is NodeJS. But frankly, as much as I like NodeJS, I nowadays try to stay away from using the standard library's streams and event emitters. Let's just say not everything in the Nodejs stdlib that *could* conceivably be used in userland software built on that foundation *should* be used.

With that off the chest, let's move on to what PipeStreams claims to provide.

[WIP]

- **The `remit` methods, `$()` and `$async()`**
- **Convenience stream transforms**
- **Circular pipelines**
- **Push sources**
- **Bridge to NodeJS streams**
- **An (optional) convention for data events**
- **Tees: diverting into multiple sinks**